L14: Entry 1 of 14 File: USPT Dec 30, 2003

DOCUMENT-IDENTIFIER: US 6671707 B1
TITLE: Method for practical concurrent copying garbage collection offering minimal thread block times

Abstract Text (1):
A method for practical concurrent copying garbage collection offering minimal thread blocking times. The method comprises achieving dynamic consistency between objects in an old memory space and objects in a new memory space. Threads are allowed to progress during garbage collection and threads are flipped one at a time. No read barrier is required.

Brief Summary Text (9):
Java was derived from the C++ programming language. Java includes some other important features from garbage collected languages (e.g., Smalltalk and LISP)--including automatic memory storage management. Garbage collected languages, such as Java, allow the system (garbage collector) to take over the burden of memory management from the programmer. When a program runs low on heap space, the garbage collector (GC) determines the set of objects that that program may still access. Objects in this set are known as live objects. The space used by objects that will no longer be accessed ("dead objects") is freed by the garbage collector for future use. An object is defined as a collection of contiguous memory locations, lying in a single region that can be addressed and accessed via references.

Brief Summary Text (12):
There are many algorithms for performing garbage collection. All the algorithms start with a set of roots that enumerate all objects in the heap that are directly reachable. A root is a slot whose referent object (if any), is considered reachable, along with all objects transitively reachable from the referent. The remaining objects in the heap are unreachable and can be reclaimed. One type of garbage collection is called conservative, or ambiguous roots, garbage collection. In conservative garbage collection, the garbage collector assumes all global variables, in registers or on the stack, are root slots even though some might hold integers, or floating point or string data. Another type of garbage collection is precise garbage collection. In precise garbage collection, the root set must unambiguously contain all reference values, or else memory errors will result. This is because precise garbage collection compacts the memory space by moving all the objects it finds to another memory region. The values in the root set must contain reference values since the garbage collector copies and moves the objects pointed to by references, and then updates the references correspondingly. If a value is mistakenly considered a reference value when it is not, a wrong piece of data will be moved, and/or a non-reference mistakenly modified, and program errors may occur.

Brief Summary Text (13):
Previous concurrent collection algorithms overlap some parts of collection with mutation, but still stop the world to "flip" (adjust, correct) all the mutator stacks and roots. A mutator thread performs application work. In a large server application, where there are perhaps hundreds of threads, thread stack flipping time can introduce unacceptable pauses.

Brief Summary Text (15):
A method for practical concurrent copying garbage collection offering minimal thread blocking times is described. The method comprises achieving dynamic consistency between objects in an old memory space and objects in a new memory space. Threads are allowed to progress during garbage collection and threads are flipped one at a time. No read barrier is required.

Drawing Description Text (7):
FIG. 4A is pseudo-code for a write barrier including the write action;

Drawing Description Text (8):
FIG. 4B is pseudo-code for the Root-Mark Phase;

Drawing Description Text (10):
FIG. 4D is code for a Copy Phase Write Barrier;

Drawing Description Text (12):
FIG. 4F is Flip Phase Write Barrier pseudo-code;

Drawing Description Text (15):
FIG. 4I is Replicate Phase Write Barrier pseudo-code; and

Detailed Description Text (4):
Existing garbage collectors stop all threads while thread stacks are adjusted to account for copied objects, or in GC parlance, the "flip" to the new copies. Some incremental or concurrent copying collectors use read barriers involving conditionals. A read barrier comprises operations performed when loading a pointer or possibly when accessing its referent object. The operations are called a barrier because the operations must be performed before the pointer use proceeds, since the barrier may replace the pointer with another one, etc.

Detailed Description Text (5):
The present enhancement does not use read barriers. The present enhancement also interferes with mutator code less since writes are less frequent than reads. Copying can have advantages over mark-sweep GC algorithms because copying allows objects to be reordered and thus reclustered to improve cache and virtual memory performance. Copying may also avoid fragmentation.

Detailed Description Text (7):
Many concurrent GC algorithms use a read barrier to synchronize collector and application activities. Read barriers tend to incur significant overhead because of the frequency of reads. The present enhancement is more practical than previous algorithms because its novel techniques do not use a read barrier. The combination of minimal blocking and no read barrier makes the present enhancement suitable to multiprocessor server applications and to many real-time systems.

Detailed Description Text (10):
Previous concurrent collection algorithms overlap some parts of collection with mutation, but still stop the world to flip all the mutator stacks and roots. In a large server application, where there are perhaps hundreds of threads, thread stack flipping time can introduce unacceptable pauses. The present enhancement may offer a solution that (a) does not stop all threads at once, since the collector can flip one thread stack at a time, and (b) minimizes the blocking time of any individual thread. A thread may have to wait to flip some, or all, of its own stack, but the thread does not wait for the collector to handle a large number of other threads. Both properties are important since the first one maintains overall throughput and the second prevents latency from varying too much.

Detailed Description Text (12):
One embodiment of the present enhancement is described with one thread performing

the collector's algorithm. Thus, on a multiprocessor with k CPUs, the multiprocessing factor for mutators may drop from k to k-1 for a time while the collector is running, but the factor does not drop to 1 as it would for a stop-the-world collector. A mutator can interact with the collector when the mutator allocates, updates heap slots, and "flips" its stack from old-space to new-space. If the mutator threads generate collector work faster than one CPU can clean up, then more CPUs can be assigned to collection work.

Detailed Description Text (20):
Using a copying collector to reorder objects can improve cache locality significantly and affect overall performance. Concurrent copying collectors need a write barrier for efficiency. The write barrier comprises operations performed when a datum (most commonly a pointer) is stored into a heap object. The operations need to be loosely synchronized with the actual update, but the synchronization requirements are generally not as stringent as for a read barrier. Generational collectors use write barriers to detect and record pointers from older to younger generations, so that upon collection the collectors can locate pointers from U (regions of memory not collected in the particular collection) to C (regions of memory collected in the particular collection) efficiently. One embodiment of the present enhancement uses more complex write barriers in some phases to bring O and N copies of objects into consistency and to assist in flipping. Some of these write barriers need to occur for all updates rather than only the updates that store pointers. The present enhancement makes a good trade-off since reads are much more common than writes, so the overall performance should be better than systems using a read barrier. Code density is also better without read barriers.

Detailed Description Text (22):
A memory region may contain slots as well as non-slot data. A slot is a memory location that may contain a pointer. For one embodiment of the present invention, three distinct regions are defined: U (Uncollected)--A region of the heap (i.e., potentially shared among all threads) whose objects are not subject to reclamation in a particular cycle of the collector. For convenience, U also includes all non-thread-specific slots not contained in objects, such as global variables of the virtual machine itself. U also includes slots managed by interfaces such as the Java Native Interface (JNI) on behalf of code external to the virtual machine. C (Collected)--A region of the heap (potentially shared among all threads) whose objects are subject to reclamation in a particular cycle of the collector. C consists only of objects and has no slots not contained within an object. C is further divided into: O (Old space)--Copies of objects as they existed when the collector cycle started. N (New space)--New copies of objects surviving the collection. S (Stack)--Each thread has a separate stack, private to that thread. S regions contain slots, but no objects, i.e., there may be no pointers from heap objects into stacks. For convenience, other thread-local slots are included into S, notably slots corresponding to those machine registers containing references.

Detailed Description Text (23):
There are two other useful things to know about the definition of U and C. First, though one might scan U to find slots referring to C, a generational system usually employs a write barrier and an auxiliary data structure, such as a remembered set of U slots that may contain pointers to C objects, to avoid scanning U. Second, during collection, new objects are not allocated in the C area; rather, the nurseries being filled during collection are considered to be part of U. This affects the write barrier used by a generational collector, or requires that the nurseries be scanned for pointers to C objects. The S and U regions contain roots, which are where collection "starts" in its determination of reachable O objects.

Detailed Description Text (24):
One embodiment is divided into two major groups of phases. The first group of phases: (a) determines which O objects are reachable from root slots in the U and S regions and (b) constructs copies of the reachable O objects in N. An object is

reachable if a root slot points to it, or a reachable object has a slot pointing to it. Reachability is the transitive closure of reference following, starting from roots. The two copies of any given reachable object are kept loosely synchronized. A synchronization point is a point in code, that when reached, entails a synchronization between threads. The Java programming language and the Java virtual machine have precise definitions of required synchronization points and their effects. The principal points are acquisition and release of monitor locks, and reads and writes of volatile variables. Any changes made by a thread to O objects between two synchronization points will be propagated to the N copies before passing the second synchronization point. This takes advantage of the Java virtual machine specification's memory synchronization rules so that updates to both copies need not be made atomically and simultaneously. If all mutator threads are at synchronization points, then the O and N copies will be consistent with one another at a particular phase of collection. This property between O and N space is called dynamic consistency.

Detailed Description Text (25):
The second group of phases is concerned with flipping S and U pointers so that the pointers point to N space and not O space. For one embodiment of the present enhancement, this group of phases uses a write barrier only (i.e., no read barrier). The present enhancement allows unflipped threads to access both O and N copies of objects, even of the same object. However, slightly tighter synchronization of updates to both copies may be required. More significantly, the present enhancement affects pointer equality comparisons (== in Java), since the system has to be able to respond that pointers to the O and N copies of the same object are equal from the viewpoint of the Java programmer. Comparing two non-null pointer values for equality is a relatively rare operation, so the extra performance cost may be marginal. Note that comparisons of pointers against null are unaffected and are likely the most frequent pointer comparisons performed in practice.

Detailed Description Text (29):
The specific early phases are: Pre-Mark, Root-Mark, Mark, Allocate, Pre-Copy, and Copy. Note that in practice a number of these phases can be combined and performed together, as described later. However, the algorithmic explanations are clearer if the phases are discussed separately and the goals and actions of each made precise.

Detailed Description Text (31):
Initially all existing objects are considered to be white. As collection proceeds, objects progress in color from white, to gray, to black. In the present enhancement, black objects are never turned back to gray and rescanned. The goal of the three marking phases (Pre-Mark, Root-Mark, and Mark) of the collector is to color every reachable C object black. Further, any object unreachable when marking begins will remain white, and the collector will reclaim it eventually. Newly allocated objects are considered gray in the pre-mark phase and black from then on.

Detailed Description Text (32):
To ensure the no-black-points-to-white rule, the mutators need to do write barrier work as described below. The marking phase write barrier ensures that the referent of any pointer stored into an object is gray or black. However, the most subtle aspect of the marking algorithm is ensuring that eventually no S slot refers to a white object.

Detailed Description Text (35):
The later mark phase requires assistance from mutator threads at their write barriers. Hence, the pre-mark phase establishes additional write barrier behavior beyond the usual generational write barrier. The pseudo-code of FIG. 4A presents a write barrier including the write action.

Detailed Description Text (36):
There are at least two ways in which this write barrier might be established. If each thread has a thread-local variable, for example a dedicated branch target register referring to the current write barrier, then all the threads are processed, updating that variable. If there is a single global variable, e.g., a state variable that is tested in a write barrier subroutine, or a single global pointer in memory referring to the current write barrier routine, then that variable or pointer can be simply be updated. Since the collector is the only thread that will update the variable in question, atomic access is not specifically required. However, the next phase cannot be started until all threads are "on-board" with the new write barrier. The gray set is initially empty before the write barrier is changed in this phase.

Detailed Description Text (37):
Conditions true at the start of the phase: All objects are white. The gray set is empty. All threads have the "standard" write barrier.

Detailed Description Text (38):
Conditions true at the end of the phase: All threads have the mark phase write barrier.

Detailed Description Text (40):
Termination: Any thread created during or after this phase starts with the appropriate write barrier. Hence only previously existing threads have to be processed, visiting each one once. This task will eventually complete. If a single global variable can be set to activate the write barrier desired, then the task consists merely of changing that variable.

Detailed Description Text (41):
2. Root-Mark Phase

Detailed Description Text (42):
This phase iterates through all U slots that could possibly refer to C objects and "grays" any white C objects referred to by those slots. The root-mark phase "blackens" the U slots. Note that as of this phase, stores into newly allocated objects, including initializing stores, have to invoke the mark-phase write barrier. Put another way, the new U slots created when objects are allocated are treated as being "black" from here on as opposed to their treatment as "gray" in the Pre-Mark phase.

Detailed Description Text (43):
While the U region can be scanned to find the relevant slots, the remembered set data structure built by a generational write barrier can be utilized to locate the relevant slots more efficiently. The pseudo-code of FIG. 4B is for the Root-Mark Phase.

Detailed Description Text (46):
Invariants of the phase: S slots are gray. All black slots are in U. Any O object grayed was reachable from a root. No objects are allocated into the O region. All threads employ the mark-phase write barrier. Black slots cannot refer to white objects.

Detailed Description Text (52):
The mark phase write barrier is applied to each slot in the object referred to by the pointer removed from the gray set. The previously gray object is now black since all its referents are gray, and any modification of the object will continue to insure that its referents are non-white. If the gray set has duplicate entries for the object, the object is considered gray until all the duplicates are processed. Put another way, gray objects are recorded explicitly, and the black

objects are simply the non-gray marked objects. To avoid scanning O later, building an explicit set of black objects may be desirable.

Detailed Description Text (53):
Marking also involves finding S pointers to O objects. At any time the collector may request a thread to scan that thread's own stack, including registers, for references to white, unmarked objects and to invoke the mark phase write barrier on them.

Detailed Description Text (54):
Scanning an individual thread's stack for pointers to white objects can be easy, but reaching the state of having no pointers to white objects in any thread stack is more difficult. This is because even after a thread's stack has been scanned, the thread can enter more white pointers into the stack since there is no read barrier preventing that from happening. The problem is using the fact that the write barrier grays a white object prior to installing in the heap any pointer to the object. For example, suppose that between a certain time t1 and a later time t2 each thread's stack has been scanned, none of the thread stacks had any white pointers, and the gray list has been empty at all times. There are now no white pointers in S or in marked O objects, and thus that marking is complete. A thread can obtain a white pointer only from a (reachable) gray or white object. There were no objects that were gray between t1 and t2, so a thread could obtain a white pointer only from a white object, and the thread must have had a pointer to that object already. But if the thread had any white pointers, the white pointers are discarded by the time the thread's stack was scanned, and thus cannot have obtained any white pointers since then. This applies to all threads, so the thread stacks cannot contain any white pointers.

Detailed Description Text (55):
The argumentation concerning reachable O objects is straightforward. The O objects initially referred to by U slots were all added to the gray set and have been processed, and since t1, the write barrier has added no additional ones. A chain of reachability from a black slot to a white object has to pass through a gray object because of the tri-color invariant. Since there are no gray objects, all reachable O objects have been marked.

Detailed Description Text (56):
The following strategies can be applied for marking. First, the collector processes the gray set until the gray set is empty. Then the collector proceeds to scan thread stacks until a stack scan adds something to the gray set. The collector then processes the gray set until the set is empty again and resumes scanning thread stacks. If the collector scans all thread stacks after the gray set becomes empty, and no items are added to the gray set by stack scanning, then marking is done. Threads that are suspended continuously since their last scan in this mark phase need not be rescanned. Not having to rescan suspended threads can be an improvement due to the presence of large numbers of threads, most of which are suspended for the short term. Likewise, if stack barriers are utilized, then old frames that have not been re-entered by a thread since the collector last scanned its stack do not have to be rescanned. (Stack barriers are described later.) Because of the possible and necessary separation of pointer stores from their associated write barriers, stack scanning appears to require that threads be brought to GC-consistent states, i.e., states where every heap store's write barrier has been executed.

Detailed Description Text (57):
Once the mark phase completes, the mark phase write barrier may be removed, though correctness is not harmed if the mark phase write barrier remains until a different write barrier is required by a later phase.

Detailed Description Text (60):
Invariants of the phase: No objects are allocated into the C region. All threads

employ the mark-phase write barrier. Black slots do not refer to white objects.

Detailed Description Text (62):
There appear to be two possible attacks on progress in marking, both resulting from the continual creation of additional threads. One attack comes if each thread comes to the same white object, decides to make the object gray, but is suspended before the graying actually happens. This condition would result in the object being entered into the gray set multiple times, with no bound on the number of times. This first attack is called the "gray set flooding attack". If a bound is imposed on the total number of threads allowed to exist at one time, then at least one of the threads will complete its write barrier and the object will no longer be added to the gray set. The maximum number of threads bounds the number of times an object can be entered. Using atomic memory operations to mark objects also avoids the gray set flooding attack. However, in practice, duplicate gray set entries should be rare and the greater cost of an atomic marking operation may not be worthwhile.

Detailed Description Text (63):
The other attack is on stack scanning. If new threads are continually created, possibly discarding old threads to stay within the maximum number imposed to avoid the gray set flooding attack, there might always be stacks not yet scanned by the collector. However, this is not really a problem. Consider the original argument and its time span from t1 to t2. Let Old be the set of threads existing at time t1 and New be threads created between time t1 and t2. If no thread in Old referred to a white object since t1, and no objects have been added to the gray set, then no thread in New can refer to a white object. For a New thread to have a pointer to a white object, the New thread would have to load the pointer from the heap since there is no direct communication between threads. All O objects reachable from U slots are black at t1. Since the gray set remained empty, that property was true from t1 to t2. That is, all reachable O slots and all U slots were black for the whole time. Thus a New thread cannot have obtained any pointers to white objects. If a New thread is created by passing arguments from an Old thread, those arguments should be blackened as part of the thread spawning process in order to ensure that white pointers cannot "leak" from Old to New threads.

Detailed Description Text (65):
The mark phases above establish which O objects are reachable. Those phases are the primary ones extended to handle Java finalization and weak pointer semantics, since those extensions to basic reachability have primarily to do with determining which objects are reachable and thus subject to copying. Once the reachable O objects are determined, an N copy is allocated for each of them during the Allocation Phase. In the Copy Phase, the O object contents are then copied to the allocated N space. The Copy Phase needs a new write barrier, to maintain dynamic consistency between the O and N copies of objects. The Pre-Copy Phase has the job of establishing that write barrier.

Detailed Description Text (75):
As object contents are copied from O space to N space, the collector needs mutator assistance to insure that updates occurring after the collector's copying operation are propagated from O versions of objects to their corresponding N versions. The mark phase write barrier is replaced with the Copy Phase Write Barrier code of FIG. 4D.

Detailed Description Text (76):
Unlike most copying collector write barriers, this write barrier applies to heap writes of non-pointer values as well as of pointers. This barrier also requires work regardless of the generational relationship of the objects in the case of storing a pointer. Finally, note that a pointer in an N object always points to U or N space, not to O space. The invariant that N objects cannot refer to an O object is maintained.

Detailed Description Text (77):
Conditions true at the start of the phase: Each black O object has a unique corresponding N copy allocated. No thread has the copy phase write barrier installed. N object contents are undefined.

Detailed Description Text (78):
Conditions true at the end of the phase: Every thread uses the copy phase write barrier.

Detailed Description Text (80):
Termination: The set of threads existing at the start of the phase is fixed and finite, and each new thread has its write barrier set appropriately as the thread is created. Thus as each thread is switched to the new write barrier a fixed set is reduced.

Detailed Description Text (83):
As the collector copies object contents, mutators may concurrently be updating the objects. The copy phase write barrier will cause the mutators to propagate their updates of O objects to the N copies, but the mutators can get into a race with the collector. To avoid making the mutator write barrier any slower or more complex than it already is, the burden of overcoming this race is placed upon the collector, as follows.

Detailed Description Text (92):
Conditions true at the end of the phase: N object contents are "dynamically consistent" with their (unique) O copies. More precisely, when no mutator is in the middle of write barrier code for a given slot, the N and O copies of that slot have consistent values. For non-pointer data, "consistent" means "equal". For pointer values, "consistent" means that the N value is the forwarded version of the O value.

Detailed Description Text (93):
Invariants of the phase: All threads use the copy phase write barrier. No new objects are allocated into the C region. All reachable O objects are black. The mapping between black O objects and their N copies is one-to-one, and onto the N copies. If an O object has an N copy, the N copy has room for the O object's data. No pointer stored into an N object refers to an O object.

Detailed Description Text (100):
The later phases for one embodiment of the present enhancement are: Pre-Flip, Heap-Flip, Thread-Flip, and Post-Flip. The goal of these phases is systematically to eliminate O pointers that may be seen and used by a thread. The plan of the phases is as follows. First, a write barrier is installed to help keep track of places possibly containing O pointers to O objects. Next, ensure that there are no heap (U region) pointers to O objects. Then start flipping threads at will.

Detailed Description Text (101):
An invariant that U and N objects do not point to O objects is established and maintained. The flip phase write barrier, installed by the Pre-Flip phase, serves to ensure that in the future no O pointers are stored into heap objects. The Heap-Flip phase eliminates any U pointers to O objects. Unflipped threads may have pointers to O and N objects, even to the same object, but flipped threads cannot hold O pointers. In the Thread-Flip phase, each flipped thread will no longer hold O pointers. The Post-Flip phase simply restores the normal (i.e., not-during-collection) write barrier and reclaims the O region.

Detailed Description Text (104):
The pre-flip phase's job is to install the Flip Phase Write Barrier. As with other write barrier installations, the installation may either be a single global operation or involve visiting each thread and doing something to the thread.

Detailed Description Text (105):
The Flip Phase Write Barrier pseudo-code is shown in FIG. 4F. The pseudo-code for implementing pointer equality tests for one embodiment is shown in FIG. 4G. This pointer equality test assumes that the thread is not suspended in the middle of the test while the collector completes collection and a new collection starts. If a thread is suspended, then the result can comprise an O version of p but a forwarded version of q, and the test could then give the wrong answer. One fix is to make sure that threads in this code advance to the end of the equality test before collection completes. Such thread advancing requirements may apply to other pseudo-code fragments described herein as well, i.e., any that examine or update forwarding pointers.

Detailed Description Text (106):
The flip-phase write barrier must be installed before the Heap-Flip phase. Otherwise unflipped threads might write O pointers in U slots. Similarly, the pointer equality test should be installed at this time, since the Heap-Flip phase will start to expose N pointers to unflipped threads.

Detailed Description Text (107):
Conditions true at the start of the phase: N object contents are dynamically consistent with their O copies. All mutator threads use the copy-phase write barrier.

Detailed Description Text (108):
Conditions true at the end of the phase: All mutators use the flip-phase write barrier. No further O pointers will be written into U objects.

Detailed Description Text (110):
Termination: There is a fixed and finite set of threads to be processed, and processing each thread takes no more than a fixed number of operations. New threads are spawned with the new write barrier, so termination is not threatened by thread creation.

Detailed Description Text (111):
2. Heap-Flip Phase

Detailed Description Text (115):
Invariants of the phase: No new objects are allocated into the C region. All reachable O objects are black, and have a unique corresponding N copy, with which they are dynamically consistent. No N object refers to an O object. No stores to U or N store an O pointer because all mutators use the flip-phase write barrier.

Detailed Description Text (118):
With the write barrier set by the pre-flip phase, flipping is straightforward. To flip a given thread, all O space references in the thread's portion of S (stack and registers) are replaced with their N space forwarded versions. This step can be done incrementally using stack barriers, as mentioned for marking. The flip-heap-pointer pseudo-code for flipping S slots can also be used. Any new threads start flipped.

Detailed Description Text (121):
Invariants of the phase: No new objects are allocated into the C region. All reachable O objects are black, and have a unique corresponding N copy, with which they are dynamically consistent. No N object refers to an O object. No stores to U or N store an O pointer because all mutators use the flip-phase write barrier.

Detailed Description Text (124):
Once all threads have been flipped, the special write barriers can be turned off and reverted back to the normal write barrier that is used when GC is not running.

The collector may then visit each N copy and remove the back pointer to its O copy, and finally, reclaim O space. The information in "fat" locks may also need to be updated if those locks include back pointers to their object. The steps of one embodiment are performed in this order: (1) change the write barrier to the normal write barrier so that threads will no longer follow back pointers to O objects; (2) after ensuring that all threads are using the new write barrier and have completed any write barriers that were in progress, remove back pointers from N objects to O objects and fix "fat" locks; (3) reclaim O space.

Detailed Description Text (125):
Conditions true at the start of the phase: N objects may have back pointers to O objects. Locks may be in "expanded" ("fat") form and shared between the N and O copies of an object. All threads use the flip phase write barrier.

Detailed Description Text (126):
Conditions true at the end of the phase: No N object has a back pointer to an O object. Locks are no longer shared between N and O copies of an object. All threads use the normal write barrier.

Detailed Description Text (130):
For one embodiment, some phases need to be strictly ordered and cannot be merged. However, a number of the earlier phases can be merged. Specifically the Root-Mark, Mark, Allocate, Pre-Copy, and Copy phases can be merged. The Pre-Mark phase necessarily precedes the new copy phase. The new copy phase is called the Replicate phase here to distinguish it from the unmerged Copy phase. The later flipping phases need to be strictly ordered or some important invariants will be violated. Since the new Pre-Mark phase installs a write barrier that is different from the old one, the new Pre-Mark phase is called the Pre-Replicate phase. This write barrier is termed the Replicate Phase Write Barrier.

Detailed Description Text (132):
The Pre-Replicate phase simply installs the Replicate Phase write barrier. This write barrier described by the pseudo-code in FIG. 4I. This write barrier simply combines the previous mark and copy phase write barriers. There are two strategies as to what add-to-gray-set does when the phases are combined. First, the mutators can do considerable work. Or second, the mutators can hand the work over to the collector. The work involved consists of allocating unique space for the newly grayed object and copying the object contents over. Having mutators do more work could avoid collector bottlenecks and share the work around on a multiprocessor. However, this strategy requires more synchronization unless N space is set up with several distinct areas into which objects can be copied (i.e., to avoid synchronization conflicts on allocation in N space). For one embodiment, mutators simply add to a list of new gray objects, and the collector does the allocation, forwarding, and copying. There can be multiple gray-object lists to reduce mutator synchronization bottlenecks. However, the collector has to then do more work to check the lists. The gray set is initially empty before the write barrier is changed in this phase.

Detailed Description Text (133):
Conditions true at the start of the phase: All objects are white. The gray set is empty. All threads have the "standard" write barrier.

Detailed Description Text (134):
Conditions true at the end of the phase: All threads have the replicate phase write barrier.

Detailed Description Text (136):
Termination: Any thread created during or after this phase starts with the appropriate write barrier. Hence, only the previously existing threads have to be worked on, visiting each thread once. This task will obviously complete.

Detailed Description Text (138):
In the replicate phase, mutators do nothing "special", except use the replicate phase write barrier. The collector acts as follows: 1. The collector scans root slots, heap slots (slots in U that might refer to O objects), and stack slots. The replicate-object code is called for each slot. The order in which slots are processed does not matter for correctness. 2. If there are any not yet scanned objects in N space, the collector calls scan-slot for unscanned object slots. 3. The collector acquires references from the gray set and calls forward-object for each reference. 4. The phase terminates when (a) all roots have been scanned, (b) all heap slots have been scanned, (c) all N objects have been scanned, and (d) all thread stack slots have been scanned while the gray object set remained empty.

Detailed Description Text (144):
Termination: The root and U slots are processed only once since the write barrier will maintain the no-black-points-to-white rule thereafter and there is a fixed number of slots at the beginning of the phase. Since O space has a fixed number of objects and slots, scanning will terminate. Each attempt to complete thread stack scanning will either complete, or gray an O object, of which there are a fixed number.

Detailed Description Text (163):
In the version of the present enhancement that merges phases, another pass of the replicate phase is performed, using the table of objects requiring finalization as a new set of roots. These objects are copied just like objects not requiring finalization. However, memory synchronization may not be necessary in the copying since only the collector can access these objects. After copying the objects, the collector adds them to the finalization thread's data structure. One simple method is for the collector to add none of the objects until after copying all of the objects since some of the unreachable objects may be reachable from other unreachable objects. However, adding the objects one at a time is legal, even though that may cause unreachable objects to become reachable. Hence memory synchronization cannot be skipped when copying the remaining objects requiring finalization or objects reachable from them.

Detailed Description Text (166):
The underlying mechanisms rely on four strengths of reachability. The strengths are: Strong reachability: This is reachability from a root via a sequence of ordinary pointers. Ordinary pointers are called "strong" in the context of finalization and weak pointers. Guarded reachability: Guarded pointers are pointers embedded in GuardedReference objects. An object is guarded-reachable if it is not strongly reachable but can be reached from a root via a sequence of pointers each of which is strong or guarded. Weak reachability: Weak pointers are pointers embedded in WeakReference objects. An object is weak-reachable if it is not strong-reachable or guarded-reachable, but is reachable from a root via a sequence of pointers each of which is strong, guarded, or weak. Phantom reachability: Phantom pointers are pointers embedded in PhantomReference objects. An object is phantom reachable if it is not strong-reachable, guarded-reachable, or weak-reachable, but is reachable from a root via a sequence of pointers each of which is strong, guarded, weak, or phantom.

Detailed Description Text (171):
A. Generational Write Barriers

Detailed Description Text (172):
In a generational collector, to avoid scanning the older generations when collecting one or more younger generations, mutator writes are tracked with a write barrier. Specifically, when object p is modified to refer to object q, that fact has to be remembered if p is in an older generation than q. Some write barrier schemes simply record something about every pointer write. For example, card

marking records the region that was modified (in the example, the region containing p or the specific slot of p that changed). Eventually, or perhaps as part of the write barrier, the information is filtered to determine if an older-to-younger pointer was created, and such pointers may be remembered across collections, etc. The important thing to note about the method of the present embodiment is that, unlike most generational schemes, the write barrier has to be applied to stores that initialize pointer fields of newly allocated objects. This requirement does not arise from the age relationships of generational collection, but rather with the fact that newly allocated objects are not placed in the C region and the collector needs to know about references to C objects from outside the C region. However, the ages of regions can be arranged as follows so that a generational write barrier will remember the pointers that need to be remembered. Make the (logical) age of the nursery older than that of the O region, so that references to O objects from nursery objects will be recorded. In order to end up with the desired remembered pointers at the end of collection, arrange for the age of the N region to be older than the nursery.

Detailed Description Text (173):
While more generational write barrier work may have to be done in the present enhancement than in a collector that includes the nurseries in every collection, ensuring termination is hard if nurseries are included in C. Also, a concurrent collector will do more total work across all CPUs than a stop-the-world collector. Hence, the present enhancement can provide minimal disruption and better system utilization.

Detailed Description Text (175):
As previously discussed, marking requires finding S pointers to O objects, i.e., scanning thread stacks. At any time the collector may request a thread to scan the thread's stack, including registers, for references to white (unmarked) objects and to invoke the mark phase write barrier on the white objects. Potentially important refinements to this process may be available.

CLAIMS:

1. A method for practical concurrent copying garbage collection offering minimal thread blocking times comprising: achieving dynamic consistency between old objects in a old memory space and new objects in a new memory space without activating a read barrier to synchronize collector and application activities during garbage collection; and flipping a first of a plurality of mutator threads to change a view for said first mutator thread from an old copy of said objects to a new copy of said objects, wherein less than all of said plurality of mutator threads are stopped while thread stacks are adjusted by said flipping, and wherein a second of said plurality of mutator threads is not blocked from concurrently executing during said flipping.

3. The method of claim 1 wherein achieving dynamic consistency comprises: installing a mark phase write barrier on a thread; scanning a root set, said root set comprising of slots and objects; determining which objects are reachable from said root slots; and marking slots and objects.

6. The method of claim 3 wherein achieving dynamic consistency further comprises: allocating space for a new copy of each reachable object; installing a copy phase write barrier; and constructing copies of said reachable objects.

7. The method of claim 1 wherein flipping pointers comprises: installing a flip phase write barrier that keeps track of memory locations possibly containing pointers to objects; scanning heap memory and fixing pointers in said heap memory pointing to old objects to refer to new copies of said old objects; and flipping threads.

9. The method of claim 7 further comprising turning off special write barriers and reverting to a normal write barrier.

10. The method of claim 9 wherein said special write barriers comprise a mark phase write barrier, a copy phase write barrier, and a flip phase write barrier.

12. A method for garbage collection comprising: scanning a root set, said root set comprising a plurality of slots and objects without enabling a read barrier to synchronize garbage collector and application activities; marking said slots and said objects; allocating space in a new memory region for new objects; copying contents of old objects to new objects; updating for a first of a plurality of mutator threads, memory references pointing to said old objects in a old memory region to refer to said new objects, wherein less than all of said plurality of mutator threads are stopped while said updating for said first mutator thread is occurring and wherein a second of said plurality of mutator threads is not blocked from concurrently executing during said updating.

16. The method of claim 12 further comprising installing a write barrier.

17. The method of claim 16 wherein said write barrier comprises a mark phase write barrier, a copy phase write barrier, and a flip phase write barrier.

18. A computer readable medium having embodied thereon a computer program, the computer program being executable by a machine to perform: achieving dynamic consistency between old objects in a old memory space and new objects in a new memory space without activating a read barrier to synchronize collector and application activities during garbage collection; and flipping pointers for a first mutator thread of a plurality of mutator threads to change a view for said first thread from an old copy of said objects to a new copy of said objects, wherein less than all of said plurality of threads are stopped while said pointers for said first mutator thread are being adjusted, and wherein a second mutator thread of said plurality of mutator threads is not blocked from executing during said pointer flipping.

20. The computer readable medium of claim 18 wherein achieving dynamic consistency comprises: installing a mark phase write barrier on a thread; scanning a root set, said root set comprising of slots and objects; determining which objects are reachable from said root slots; and marking slots and objects.

22. The computer readable medium of claim 20 wherein achieving dynamic consistency further comprises: allocating space for a new copy of each reachable object; installing a copy phase write barrier; and constructing copies of said reachable objects.

23. The computer readable medium of claim 18 wherein flipping pointers comprises: installing a flip phase write barrier that keeps track of memory locations possibly containing pointers to objects; scanning heap memory and fixing pointers in said heap memory pointing to old objects to refer to new copies of said old objects; and flipping threads.

24. The computer readable medium of claim 18 further comprising turning off special write barriers and reverting to a normal write barrier, said special write barriers comprising a mark phase write barrier, a copy phase write barrier, and a flip phase write barrier.

25. A digital processing system having a processor operable to perform: achieving dynamic consistency between old objects in a old memory space and corresponding new objects in a new memory space without activating a read barrier to synchronize collector and application activities during garbage collection; and flipping pointers for a first application thread referring to said old objects to refer to

said corresponding new objects, wherein less than all application threads of said system are stopped during garbage collection, and wherein at least one of said application threads is not blocked from executing during said pointer flipping.

26. The digital processing system of claim 25 wherein achieving dynamic consistency comprises: installing a mark phase write barrier on a thread; scanning a root set, said root set comprising of slots and objects; determining which objects are reachable from said root slots; and marking slots and objects.

27. The digital processing system of claim 26 wherein achieving dynamic consistency further comprises: allocating space for a new copy of each reachable object; installing a copy phase write barrier; and constructing copies of said reachable objects.

28. The digital processing system of claim 25 wherein flipping pointers comprises: installing a flip phase write barrier that keeps track of memory locations possibly containing pointers to objects; scanning heap memory and fixing pointers in said heap memory pointing to old objects to refer to new copies of said old objects; and flipping threads.

29. The digital processing system of claim 25 further comprising turning off special write barriers and reverting to a normal write barrier, said special write barriers comprising a mark phase write barrier, a copy phase write barrier, and a flip phase write barrier.